PARALLELIZATION OF OPTIMIZERS FOR SINGLE OBJECTIVE FUNCTIONS

J Component Project Report for the course

CSE4001 Parallel and Distributed Computing

by

HARIHARAN S (18BCE1066)
MOHIT SAI ARAVIND (18BCE1319)
SHIVA MANICKAM M (18BCE1272)

Submitted to

Prof. Ayesha SK

SCHOOL OF COMPUTER SCIENCE
AND ENGINEERING
VELLORE INSTITUTE OF TECHNOLOGY
CHENNAI - 600127

November 2020

Certificate

This is to certify that the Project work titled “PARALLELIZATION OF OPTIMIZERS FOR SINGLE OBJECTIVE FUNCTIONS” is being submitted by ***Hariharan S*** **(18BCE1066),** ***Mohit Sai Aravind (18BCE1319), Shiva Manickam M (18BCE1272)*** for the course IoT Fundamentals, is a record of bona fide work done under my guidance. The contents of this project work, in full or in parts, have neither been taken from any other source nor have been submitted to any other Institute or University.

Prof. Ayesha SK

ABSTRACT

Single objective optimization algorithms are the basis of the more complex optimization algorithms such as multi-objective, niching, dynamic, constrained optimization algorithms and so on. Research on single objective optimization algorithms influence the development of the optimization branches mentioned above. In the recent years, various kinds of novel optimization algorithms have been proposed to solve real-parameter optimization problems. Our project aims to parallelize such optimizers for solving single objective problems with the help of threading libraries. Evaluation of the performance of these optimizers on specific problems can be studied under different constraints as a part of a larger framework, to improve performance of the algorithms and better understand the suitability of the same.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Optimization refers to the process of finding the best possible solutions for a particular problem. As the complexity of problems increases, over the last few decades, the need for new optimization techniques becomes evident more than before. Mathematical optimization techniques used to be the only tools for optimizing problems before the proposal of heuristic optimization techniques. Mathematical optimization methods are mostly deterministic that suffer from one major problem: local optima entrapment. Some of them such as gradient-based algorithms require derivation of the search space as well. This makes them highly inefficient in solving real problems.

Parallelization of each of the optimizers: moth-flame, whale, grey wolf optimizer will be done using the openMP threading library. Implementation and evaluation of these optimizers and the corresponding performance of the parallelized version will be tested with the help of the JMetalCPP framework.

Keywords: *Moth-Flame optimizer, Whale optimizer, Grey wolf optimizer, single objective optimization, metaheuristic optimization*

1.1 Related Work:

Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm, Seyedali Mirjalili

In this paper a novel nature-inspired optimization paradigm is proposed called Moth-Flame Optimization (MFO) algorithm. The main inspiration of this optimizer is the navigation method of moths in nature called transverse orientation. Moths fly in night by maintaining a fixed angle with respect to the moon, a very effective mechanism for travelling in a straight line for long distances. However, these fancy insects are trapped in a useless/deadly spiral path around artificial lights. This paper mathematically models this behaviour to perform optimization. The MFO algorithm is compared with other well-known nature-inspired algorithms on 29 benchmark and 7 real engineering problems

Whale optimization problem
The paper lays the foundation for the whole project and most related Whale optimization[1] technique problems. This paper talks about the possibility of mimicking the Humpback whales' hunting method. This is the first proposal of the base idea (WOA).

The paper involves improvements in Whale Optimization. One Elite Opposition-Based Learning (EOBL) at the initialization phase of WOA. Other Incorporation of evolutionary operators from the Differential Evolution algorithm at the end of each WOA [2]. Thus, this proves that there is a performance improvement against standard meta-heuristic algorithms

The paper Implements the training process of Neural Networks using Whale Optimization Algorithm [3]. Its conclusions are highly dependable as the algorithm is rigorously tested with different levels of data. Proves that WOA outperforms many standard algorithms in terms of both local optima avoidance and convergence speed.

[1] Seyedali Mirjalili, Andrew Lewis, The Whale Optimization Algorithm, Advances in Engineering Software, Volume 95, 2016, Pages 51-67, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2016.01.008.

[2] Tubishat, M., Abushariah, M.A.M., Idris, N. et al. Improved whale optimization algorithm for feature selection in Arabic sentiment analysis. Appl Intell 49, 1688–1707 (2019). https://doi.org/10.1007/s10489-018-1334-8

[3] Aljarah, I., Faris, H. & Mirjalili, S. Optimizing connection weights in neural networks using the whale optimization algorithm. Soft Comput 22, 1–15 (2018). https://doi.org/10.1007/s00500-016-2442-1

Grey wolf optimization problem, Seyedali Mirjalili

The work proposes a new meta-heuristic called Grey Wolf Optimizer (GWO) inspired by grey wolves (Canis lupus). The GWO algorithm mimics the leadership hierarchy and hunting mechanism of greywolves in nature. Four types of grey wolves such as alpha, beta, delta, and omega are employed for simulating the leadership hierarchy. In addition, the three main steps of hunting, searching for prey, encircling prey, and attacking prey, are implemented. The algorithm is then benchmarked on 29 well-known test functions, and the results are verified by a comparative study with Particle Swarm Optimization (PSO), Gravitational Search Algorithm (GSA), Differential Evolution (DE), Evolutionary Programming (EP), and Evolution Strategy (ES). The results show that the GWO algorithm is able to provide very competitive results compared to these well-known meta-heuristics

Optimization Algorithms

To solve the optimization problem, efficient search or optimization algorithms are needed. There are many optimization algorithms which can be classified in many ways, depending on the focus and characteristics.

If the derivative or gradient of a function is the focus, optimization can be classified into gradient-based algorithms and derivative-free or gradient-free algorithms. Gradient-based algorithms such as hill-climbing use derivative information, and they are often very efficient. Derivative-free algorithms do not use any derivative information but the values of the function itself. Some functions may have discontinuities or it may be expensive to calculate derivatives accurately, and thus derivative-free algorithms such as Nelder-Mead downhill simplex become very useful

.

From a different perspective, optimization algorithms can be classified into trajectory-based and population-based. A trajectory-based algorithm typically uses a single agent or one solution at a time, which will trace out a path as the iterations continue. Hill-climbing is trajectory-based, and it links the starting point with the final point via a piecewise zigzag path.

Another important example is simulated annealing which is a widely used metaheuristic algorithm. On the other hand, population-based algorithms such as particle swarm optimization (PSO) use multiple agents which will interact and trace out multiple paths (Kennedy and Eberhardt, 1995).

Optimization algorithms can also be classified as deterministic or stochastic. If an algorithm works in a mechanical deterministic manner without any random nature, it is called deterministic. For such an algorithm, it will reach the same final solution if we start with the same initial point. Hill-climbing and downhill simplex are good examples of deterministic algorithms.

On the other hand, if there is some randomness in the algorithm, the algorithm will usually reach a different point every time the algorithm is executed, even though the same initial point is used. Genetic algorithms and PSO are good examples of stochastic algorithms. Search capability can also be a basis for algorithm classification. In this case, algorithms can be divided into local and global search algorithms.

Local search algorithms typically converge towards a local optimum, not necessarily (often not) the global optimum, and such an algorithm is often deterministic and has no ability to escape from local optima. Simple hill-climbing is such an example. On the other hand, for global optimization, local search algorithms are not suitable, and global search algorithms should be used.

Modern metaheuristic algorithms in most cases tend to be suitable for global optimization, though not always successful or efficient. A simple strategy such as hill-climbing with random restarts can turn a local search algorithm into an algorithm with global search capability. In essence, randomization is an efficient component for global search algorithms.

OpenMP
OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures. It provides various constructs and directives for specifying parallel regions, work sharing, synchronization and data environment. Runtime library functions and environment variables are also available.

JMetalCPP

The jMetalCPP project is a C++ based easy to use, flexible, extensible and portable multi-objective optimization framework with metaheuristics.
Features of JMetalCPP:

- Architecture redesign to provide a simpler design while keeping the same functionality.
- Maven is used as the tool for development, testing, packaging and deployment.
- Promote code reusing by providing algorithm templates
- Improve code quality:
    - Application of unit testing
    - Better use of Java features (e.g, generics)
    - Design patterns (singleton, builder, factory, observer)
- Parallelism support
- Introducing measures to get information of the algorithms in runtime

1.2 Objectives

The aim of the project is to perform a comprehensive study on the three single objective metaheuristic optimizers: moth-flame, whale, grey wolf optimizer. The various phases of the project will try to achieve the following objectives:

1. Definition of the single objective problem
2. Implementation of each of the optimizers
3. Modular design for every problem and optimizer implemented
4. Parallelization of the optimizer functions with the help of threading libraries such as openMP
5. Evaluation and benchmark tests for each of the optimizers

1.3 Scope

The project will aim to parallelize only the functions that exist as a part of the original optimizer, and will not modify the working of the optimizer as such. A threaded implementation will simply parallelize the execution and make the program run more efficiently. Suitable tests and benchmarks will be tested for the parallelized optimizers. The difference in time taken is considered and all the tests are performed in a single machine for accurate results.

2. DESIGN/IMPLEMENTATION

2.1 Design of Optimizer and Problem

Moth-Flame Optimizer

M is a matrix that contains the positions of n moths in d dimensions
Fitness values for moth are stored in OM
Fitness value for flames is stored in F
The system follows a logarithmic spiral path and the distance D is calculated and updated every instant
r and t are the random number generated and time respectively

These parameters are used as per the algorithm:

Update flame no using $flame\ no = round\left(N - l * \frac{N-1}{T}\right)$
OM = FitnessFunction(M);
if iteration == 1
    F = sort(M);
    OF = sort(OM);
else
    F = sort(M_{t-1}, M_t);
    OF = sort(M_{t-1}, M_t);
end
for i = 1: n
    for j = 1: d
        Update r and t
        Calculate D using $D_i = |F_j - M_i|$ respect to the corresponding moth
        Update M(i,j) using $S(M_i, F_j) = D_i \cdot e^{bt} \cdot \cos(2\pi t) + F_j$ respect to the corresponding moth
    end
end

Figure 1: Moth flame optimizer algorithm

Whale Optimizer

The WOA algorithm follows the standard design as in Whale optimization with the following changes, the search agent jobs are run in parallel with the same number of threads as that of the system processors. This includes Fitness Calculation and Position update process. These are the major computation parts that can be parallelized.

1. Random Initialize whale population, position and leader score
2. Calculate the fitness of different members, set the first leader (Parallel)
3. Update the position of the current search agent by the default equations and recalculate fitness (Parallel)
4. Check all the search agents and update X* and other parameters if there is a better solution
5. Repeat 2 to 4 for the given number of iterations
6. Return the best result till now

Grey wolf optimizer

Initialize the grey wolf population X_i (i = 1, 2, ..., n)
Initialize a, A, and C
Calculate the fitness of each search agent
X_α=the best search agent
X_β=the second best search agent
X_δ=the third best search agent
while *(t < Max number of iterations)*
  for *each search agent*
    Update the position of the current search agent by $\vec{x}_{(t+1)}=\frac{\vec{X}_1+\vec{X}_2+\vec{X}_3}{3}$
  end for
  Update a, A, and C
  Calculate the fitness of all search agents
  Update X_α, X_β, and X_δ
  t=t+1
end while
return X_α

Figure 2: Grey wolf optimizer algorithm

2.2 Software Implementation

The following codes demonstrate the parallelization of the optimizers:s

Moth Flame Optimizer

```
SolutionSet * MFO::execute() {
      int FlameNo;
      double a;

      #pragma omp parallel for num_threads(4)
      for(int iteration=0; iteration<maxIterations_; iteration++) {
            calculateFitness();

            if(iteration == 0) {
                  for(int agentIndex=0; agentIndex<searchAgentsCount_;
agentIndex++) {
                        bestFlames_->add(new Solution(positions_-
>get(agentIndex)));
                  }

                  // Sort the first population of moths
                  bestFlames_->sort(comparator);
            } else {
                  // Sort the moths
                  doublePopulation_ = previousPositions_-
>join(bestFlames_);
                  doublePopulation_->sort(comparator);
```

```
                  // Update the flames
                  for(int agentIndex=0; agentIndex<searchAgentsCount_;
agentIndex++) {
                        bestFlames_->replace(agentIndex, new
Solution(doublePopulation_->get(agentIndex)));
                  }

                  delete doublePopulation_;
            }

            // Update the position of best flame obtained so far
            #pragma omp parallel for num_threads(2)
            for(int agentIndex=0; agentIndex<searchAgentsCount_;
agentIndex++) {
                  previousPositions_->replace(agentIndex, new
Solution(positions_->get(agentIndex)));
            }

            // a linearly dicreases from -1 to -2 to calculate t in Eq.
(3.12)
            a = -1.0 + (double)(iteration + 1) * (-1.0 /
(double)maxIterations_);

            FlameNo = round((searchAgentsCount_ - 1) - (iteration + 1) *
                              ( (double)(searchAgentsCount_ - 1) /
(double)maxIterations_));

            updatePosition(a, FlameNo);

            convergenceCurve_[iteration] = bestFlames_->get(0)-
>getObjective(0);
      }

      // Return best flame
      Solution *fitness = new Solution(bestFlames_->get(0));
      problem_->evaluate(fitness);

      SolutionSet *resultPopulation = new SolutionSet(1);
      resultPopulation->add(fitness);

      return resultPopulation;
} // execute
```

Whale Optimizer

```
SolutionSet * WOAParallel::execute() {
      double temp;
      for(int iteration=0; iteration<maxIterations_; iteration++) {
            //---------- CALCULATE FITNESS ------------
            #pragma omp parallel for num_threads(2)
            for(int agentIndex=0; agentIndex<searchAgentsCount_;
agentIndex++) {
                  adjustBoundries(agentIndex);

                  Solution *fitness = new Solution(positions_-
>get(agentIndex));
                  problem_->evaluate(fitness);
                  double fitnessValue = fitness->getObjective(0);
                  #pragma omp critical
```

```
                  {
                        if(fitnessValue < leaderScore_) {
                              leaderScore_ = fitnessValue;
                              leaderPos_ = positions_->get(agentIndex);
                        }
                  }
                  delete fitness;
            }

            double a = 2.0 - (double)iteration * ( 2.0 / maxIterations_);
            double a2 = -1.0 + (double)iteration * ( -1.0 /
maxIterations_);

            //---------- UPDATE POSITION -------------
            #pragma omp parallel for num_threads(2)
            for(int agentIndex=0; agentIndex<searchAgentsCount_;
agentIndex++) {
                  double A = 2.0 * a * PseudoRandom::randDouble(0.0, 1.0) -
a;
                  double C = 2.0 * PseudoRandom::randDouble(0.0, 1.0);
                  double b = 1.0;
                  double l = (a2 - 1.0) * PseudoRandom::randDouble(0.0,
1.0) + 1.0;
                  double p = PseudoRandom::randDouble(0.0, 1.0);

                  XReal *solutionVariables = new XReal(positions_-
>get(agentIndex));
                  //#pragma omp parallel for num_threads(2)
                  for(int variable=0; variable<solutionVariables->size();
variable++) {
                        if(p < 0.5) {
                              if(fabs(A) >= 1.0) {
                                    int randomLeaderIndex =
PseudoRandom::randInt(0, searchAgentsCount_ - 1);
                                    XReal *randomPosition = new
XReal(positions_->get(randomLeaderIndex));
                                    double D_X_rand = fabs(C *
randomPosition->getValue(variable) - solutionVariables-
>getValue(variable) );
                                    solutionVariables->setValue(variable,
randomPosition->getValue(variable) - A * D_X_rand);
                                    delete randomPosition;
                              } else {
                                    XReal *leaderPosition = new
XReal(leaderPos_);
                                    double D_Leader = fabs(C *
leaderPosition->getValue(variable) - solutionVariables-
>getValue(variable));
                                    solutionVariables->setValue(variable,
leaderPosition->getValue(variable) - A * D_Leader);
                                    delete leaderPosition;
                              }
                        } else {
                              XReal *leaderPosition = new
XReal(leaderPos_);
                              double distanceToLeader =
fabs(leaderPosition->getValue(variable) - solutionVariables-
>getValue(variable));
                              solutionVariables->setValue(variable,
distanceToLeader * exp(b * l) * cos(l * 2.0 * M_PI) + leaderPosition-
>getValue(variable));
```

```
                        delete leaderPosition;
                    }
                }
                delete solutionVariables;
            }
            convergenceCurve_[iteration] = leaderScore_;
      }

      Solution *fitness = new Solution(leaderPos_);
      problem_->evaluate(fitness);
      leaderScore_ = fitness->getObjective(0);

      SolutionSet *resultPopulation = new SolutionSet(1);
      resultPopulation->add(fitness);

      return resultPopulation;
} // execute
```

Grey Wolf Optimizer

```
SolutionSet * GWO::execute() {
      #pragma omp parallel for
      for(int l=0; l<maxIterations_; l++) {
            calculateFitness();
            double a = 2.0 - (double)l * (2.0/maxIterations_);
            updateWolves(a);
            convergenceCurve_[l] = alphaScore_;
      }

      Solution *fitness = new Solution(alphaPosition_);
      problem_->evaluate(fitness);

      // Return alpha vector
      SolutionSet *resultPopulation = new SolutionSet(1);
      resultPopulation->add(fitness);

      return resultPopulation;
} // execute
```

3. RESULTS

The optimizers were tested for the standard single objective sphere problem for comparison of performance. Multiple trials of various tests were conducted for better accuracy. The specifications of the system in which the tests were conducted is:

System Configuration: (Virtual Machine)
OS: Kali Linux 20.2, Debian (64bit)
Memory System: 4096MB
Memory BIOS: 128KB
Processors: Intel Core i7-7700HQ CPU @ 2.80GHz x 2

The following figures show the execution and the time taken for each of the optimizers on the standard sphere problem:

```
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ls
Compare  MFO_main  WOA_main  WOAParallel_main
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.216792s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.57076s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.313135s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.217655s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.568222s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.342164s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.21793s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.57236s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.365573s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.241392s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.582345s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.315514s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.21851s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.575017s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.343143s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$ ./Compare
Default problem Sphere is used
Whale Optimizer Parallel Execution
Total execution time: 0.243311s
Moth Flame Optimizer Parallel Execution
Total execution time: 0.666426s
Grey Wolf Optimizer Parallel Execution
Total execution time: 0.325383s
abckali@abckali:~/Desktop/PDC/jMetalCpp-master/bin/main$
```

Figure 3: testing parallelized optimizer trial 1 to 6

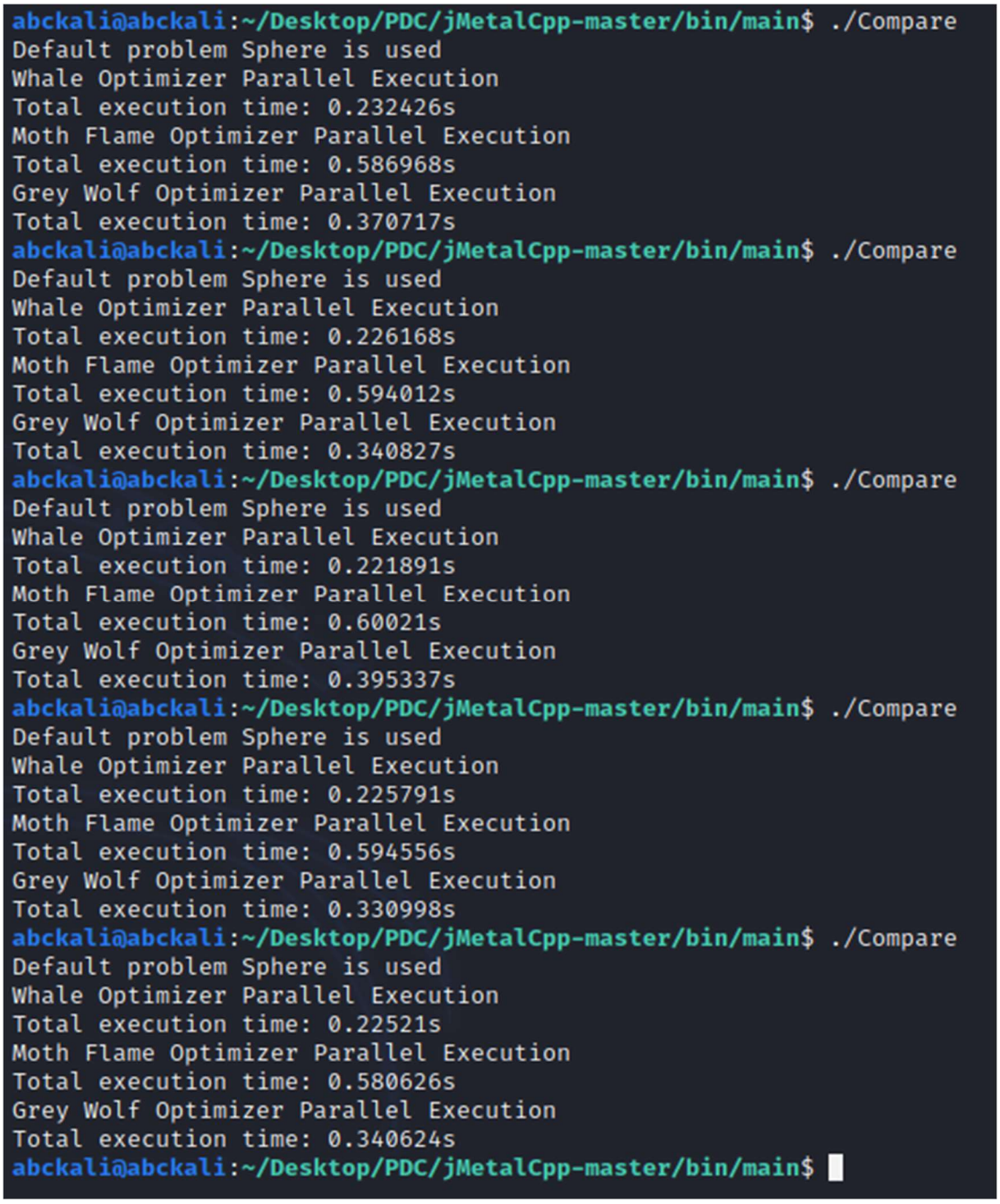

Figure 4: testing parallelized optimizer trial 7 to 10

The results of the tests are summarized in the following table:

TRIAL	WOA	MFO	GWO
1	0.2324	0.5869	0.3707
2	0.2261	0.5940	0.3408
3	0.2219	0.6002	0.3953

4	0.2258	0.5945	0.3309
5	0.2251	0.5806	0.3406
6	0.2168	0.5707	0.3131
7	0.2176	0.5682	0.3421
8	0.2197	0.5723	0.3655
9	0.2414	0.5823	0.3155
10	0.2185	0.5750	0.3431

Table 1: Comparison of time taken for each optimizer on sphere problem

The graph for time taken in each trial for the optimizers:

Legend:

COLOR	RED	GREEN	BLACK
OPTIMIZER	WHALE	GREY WOLF	MOTH FLAME

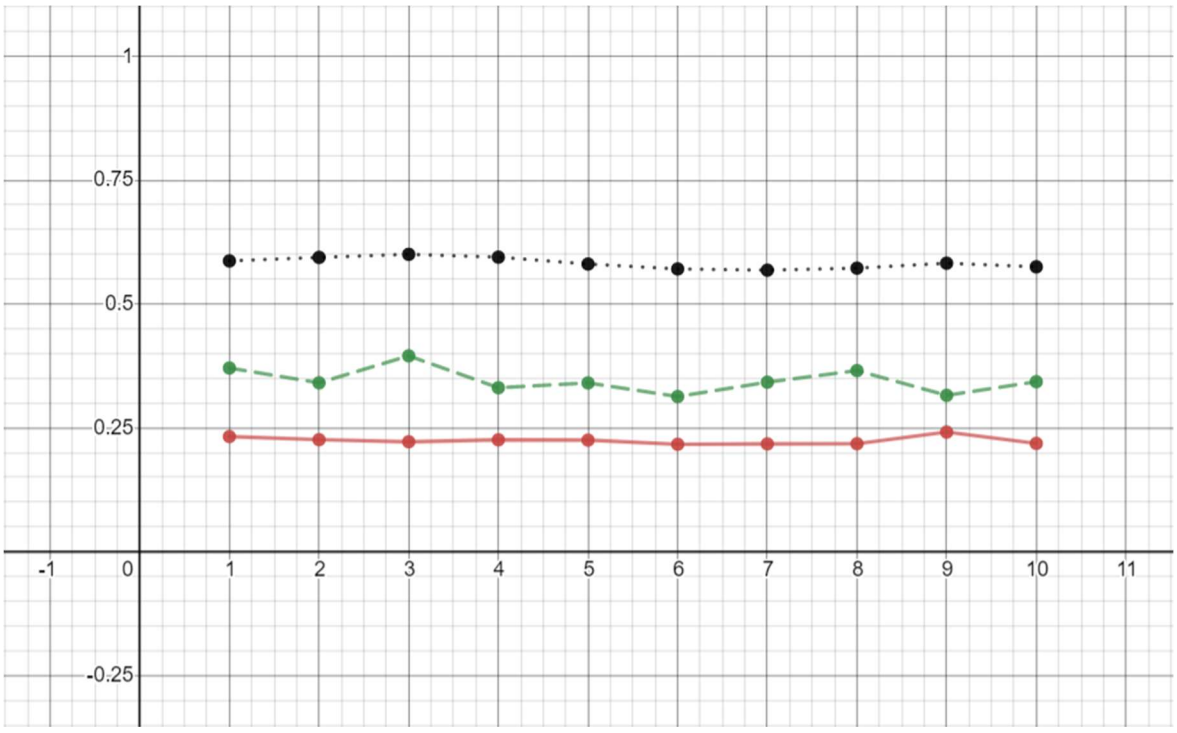

Figure 5: Graph for comparison of time taken by each optimizer

From our parallel models of the three new single objective meta-heuristic optimization algorithms, Moth flame, Whale and Grey Wolf, Whale optimization performs the best on a single objective standard sphere problem followed closely by Grey Wolf while Moth flame

performs worst. All are run with the same problem set and same parameters such as Maximum Iterations and Search agent counts

4. CONCLUSION AND FUTURE WORK

Various kind of optimizers can be used to solve metaheuristic problems. moth-flame, grey wolf and whale optimizers attempt to solve single objective/ multiple objective problems by replicating processes in nature. The study has shown that there is a significant advantage of parallelizing such optimizers. Whale optimization performs the best on certain standard single objective problems, followed by grey wolf and moth flame. Further work can be done towards redesigning the algorithm, adding or removing parameters, and specifying constraints for better suitability for parallel computing.

5. REFERENCES

[1] Seyedali Mirjalili, Andrew Lewis, The Whale Optimization Algorithm, Advances in Engineering Software, Volume 95, 2016, Pages 51-67, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2016.01.008.

[2]Tubishat, M., Abushariah, M.A.M., Idris, N. et al. Improved whale optimization algorithm for feature selection in Arabic sentiment analysis. Appl Intell 49, 1688–1707 (2019). https://doi.org/10.1007/s10489-018-1334-8

[3]Aljarah, I., Faris, H. & Mirjalili, S. Optimizing connection weights in neural networks using the whale optimization algorithm. Soft Comput 22, 1–15 (2018). https://doi.org/10.1007/s00500-016-2442-1

[4]Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm, Seyedali Mirjalili

[5] Grey Wolf Optimizer Seyedali Mirjalili , Seyed Mohammad Mirjalili , Andrew Lewis